Aleksas Murauskas 260718389
Florence Diep 26072717117
ECSE 416 - Telecommunications Networks
September, 28th 2020

Experiment 1 Report - A Basic Web server

1. Introduction

The goal of this lab is to implement a server and client via socket programming in Python. The server must process the client's HTTP request for a certain file, find the requested file within its file system, and then send an HTTP response containing the file's content over TCP connection. Secondly, the client program should be able to print to the terminal the connection status, a query acknowledgement, the received HTTP message and the content type. Initially, the client must take in the host name, port number, desired file, and an optional timeout timer as arguments as the main requirement for the server. The program should be able to process text (.txt) and image (.jpg) data. In the event that the user is requesting an invalid file (e.g. invalid input or nonexistent file), the server must send back an HTTP 404 response error message. Finally, both applications must also be able to close the sockets.

2. Web Server & Client Program Design

The server program begins by creating a socket bound to a constant server name and port number, in our case, 127.0.0.2 and 12345. The server then activates the socket to listen, and then enters an infinite loop to work through client requests. When the server connects with a client, it first expects a HTTP request that contains the requested filename. The server uses a try-except statement in order to catch an IOerror that occurs when the server attempts to open a file that does not exist. In the case of an non-existent file, the server sends a HTTP 404 Error message, then closes the socket and continues the loop to await the next request. If the requested file is a .txt or .jpg file, the program will store the data and record the MIME type. Otherwise, the server will note the invalid file type and report an HTTP 404 error to the client. If the file exists, the HTTP 200 OK response is sent to the client, followed by the MIME type, followed by the file's contents. Afterwards, the server concludes the connection by closing the socket and returns to the top of the loop in order to facilitate another request.

The client begins by parsing the arguments passed when it is run. The program expects four or 5 arguments, otherwise the program notifies the user for the incorrect number of arguments and terminates. The program intakes these values as (in order) the server name, port number, filename, and an optional timeout value. The client then creates a socket set with the desired timeout. If no timeout is input, a timeout of 5 seconds is default. The socket then tries to connect to the desired server. If the inputs are wrong or the server is offline, the program notifies the user of its inability to connect and terminates. If the connection is ok, then the client sends a request for a certain file and awaits a response from the server. If it is a 404 error, the program prints the 404, closes the socket and then terminates. If no 404 error is received, the client awaits the content type from the server to then print the type and show the content. The client program concludes closing the socket and terminating.

Content Type Handling

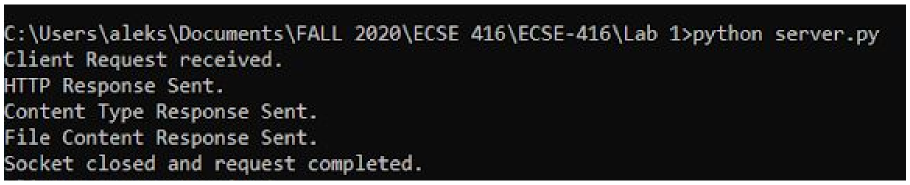
Before processing the content, the server checks if the filename ends with either ".txt" or ".jpg". Handling text data is relatively simple, but image data is much more complex. For images, we have used the pillow module as the os module cannot simply "read" an image unlike for text files, but the file content is not received in a tangible format on the client side. As such, we have decided to use the `pickle` module as it will allow us to easily serialize and deserialize any objects.

In order to ensure that the file content can be sent properly to the client, pickling allows us to convert the file object into a byte stream and send that over to the client in a cleaner format, regardless of the mime type. The client can then easily invert the operation by unpickling to turn the bytes back into its original state as an object. However, prior to unpickling, if the size of the pickled data is too big, the data becomes truncated. To accommodate any size, we created an infinite loop on the client side to gather the pickled file data in sizes of 1024 bits at a time, append them together until it receives an empty packet and unpickle the received data back into its original file content.

To display the file content, the approach used to distinguish file types in the server side must also be used in the client side. The server will tell the client via `mimeTypeResponse` message to determine the content type, print the file content to the command line interface and display accordingly. If the response is simply "text/html", the program will simply print. If it's an image, the method `show()` is used instead for visualization purposes.

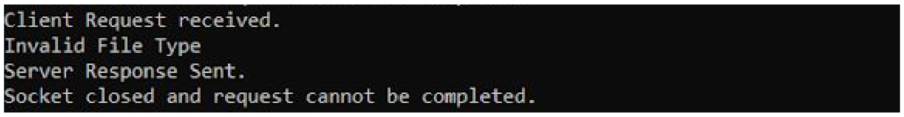
3. Testing

Throughout the design of the server, we would test at each requirement by simply printing a status message to the terminal. In total, there are three different types of messages that our server outputs to the terminal. For instance, to ensure that a connection socket was created, figure 3.1 shows the terminal output. We would do a similar approach for every communication instance between the server and the client. This proved to be very helpful when debugging our code. For instance, should the MIME type be sent from the server but an error occurs on the client side, then something must've happened around the encoding instance. Similarly, we would also include error responses from the server in the event that an unknown file or invalid file be requested. Another helpful technique was to print the raw output of the file content on the client side. This allowed us to understand that serialization of the content was required as it showed that the server was encoding the content in an irreversible format. Below are more examples of how we tested our applications.



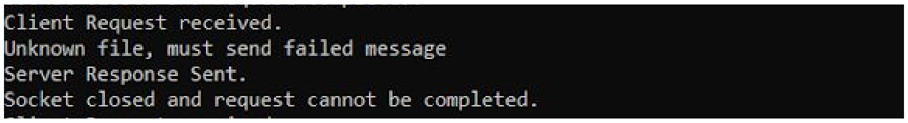
```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python server.py
Client Request received.
HTTP Response Sent.
Content Type Response Sent.
File Content Response Sent.
Socket closed and request completed.
```

Figure 3.1 Successful socket connection



```
Client Request received.
Invalid File Type
Server Response Sent.
Socket closed and request cannot be completed.
```

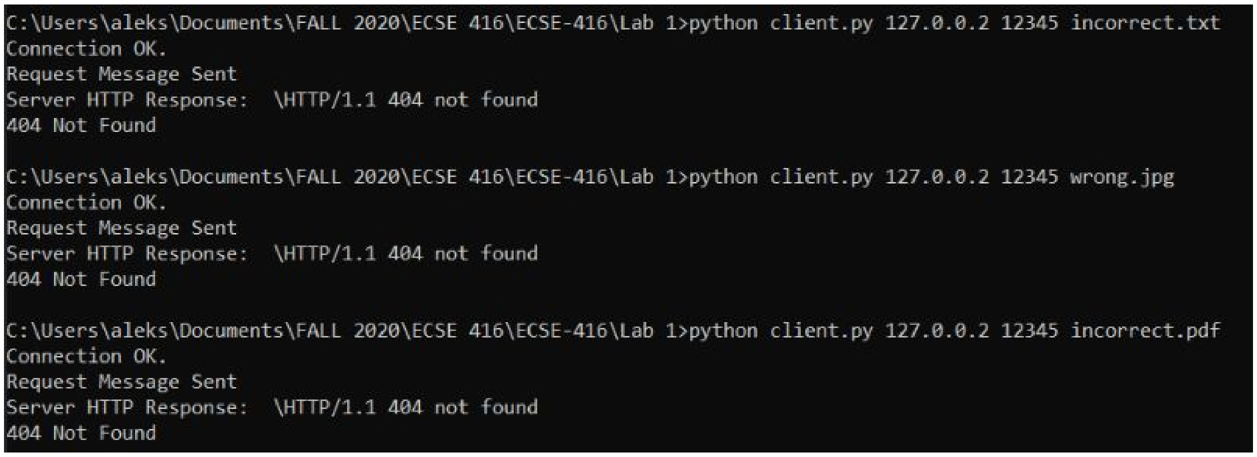
Figure 3.2 Invalid file MIMEType response



```
Client Request received.
Unknown file, must send failed message
Server Response Sent.
Socket closed and request cannot be completed.
```

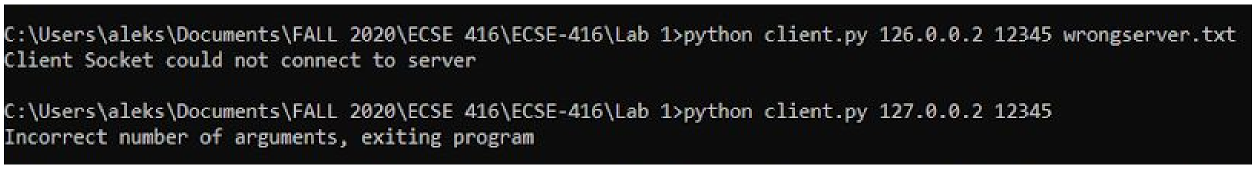
Figure 3.3 Unknown file NAME response

Similarly, in order to ensure our program was working correctly, we tested our client through a large number of situations. In addition to doing a similar approach as in the server side, We wanted to make sure that the server was returning the appropriate error message despite having the basic functionalities fulfilled. Hence, the three main tests were on retrieving an unrecognizable file, the number of arguments, and connecting to an incorrect server number. As seen in figure 3.4, we can see that the client has sent an invalid file, both in terms of filename and type. It is important to notice here that although an 404 error message was received, the connection still went through and is displayed, as desired. In regards to the number of valid inputs, the process would stop at the client side as we wouldn't want to create a connection with the server with incorrect arguments. In figure 3.5, we can see that the client side error message would depend on the invalidity of server information or inefficient number of arguments. Lastly, in figure 3.6, we test the timeout. We can see that in both scenarios, the timeout works regardless of the success of the connection.



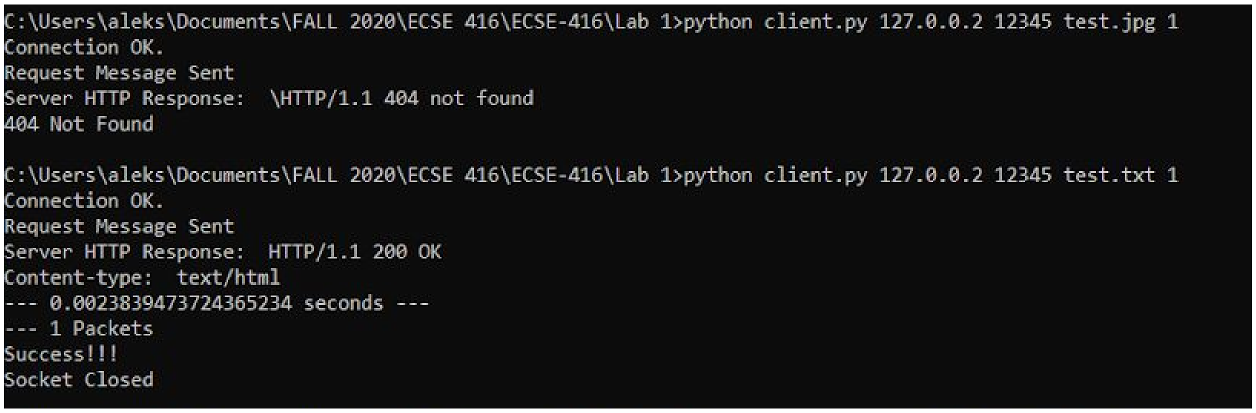
```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 incorrect.txt
Connection OK.
Request Message Sent
Server HTTP Response:  \HTTP/1.1 404 not found
404 Not Found

C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 wrong.jpg
Connection OK.
Request Message Sent
Server HTTP Response:  \HTTP/1.1 404 not found
404 Not Found

C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 incorrect.pdf
Connection OK.
Request Message Sent
Server HTTP Response:  \HTTP/1.1 404 not found
404 Not Found
```

Figure 3.4 Invalid files server response



```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 126.0.0.2 12345 wrongserver.txt
Client Socket could not connect to server

C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345
Incorrect number of arguments, exiting program
```

Figure 3.5 Invalid socket and arguments server response



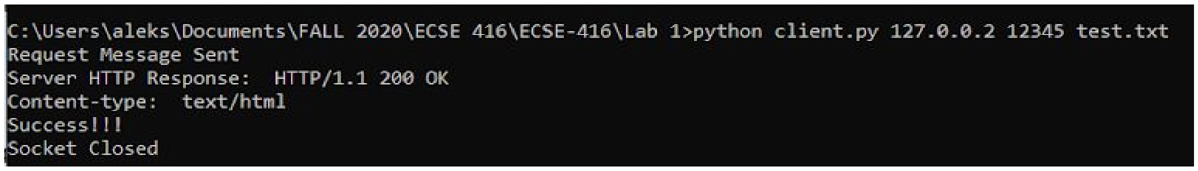
```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 test.jpg 1
Connection OK.
Request Message Sent
Server HTTP Response:  \HTTP/1.1 404 not found
404 Not Found

C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 test.txt 1
Connection OK.
Request Message Sent
Server HTTP Response:  HTTP/1.1 200 OK
Content-type:  text/html
--- 0.0023839473724365234 seconds ---
--- 1 Packets
Success!!!
Socket Closed
```

Figure 3.6 Timeout Response

4. Experiments

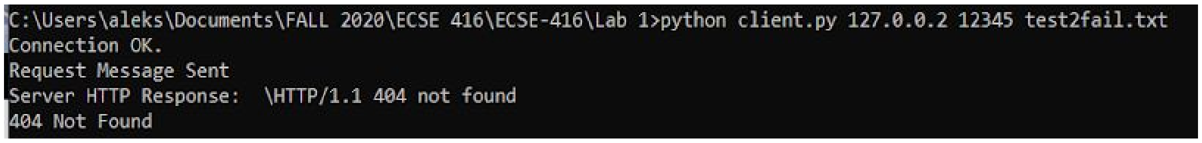
Experiment 1

The results where the client asks for a file that exists within the directory can be seen in the Figure 4.a. The server notifies the client that the file is found, the content is sent to the client, which the client immediately prints to console. To retrieve a file that does not exist, those results can be seen in Figure 4.b. The client program is notified of a 404 error attempting to find the requested file. The client prints this error and then the program terminates.



```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 test.txt
Request Message Sent
Server HTTP Response:  HTTP/1.1 200 OK
Content-type:  text/html
Success!!!
Socket Closed
```

Figure 4.a: Test 1, client asks server for an existing text file

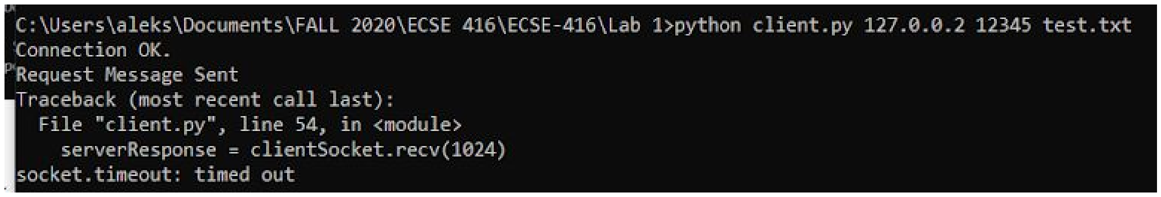


```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 test2fail.txt
Connection OK.
Request Message Sent
Server HTTP Response:  \HTTP/1.1 404 not found
404 Not Found
```

Figure 4.b: Test 2, client asks server for an nonexistent text file

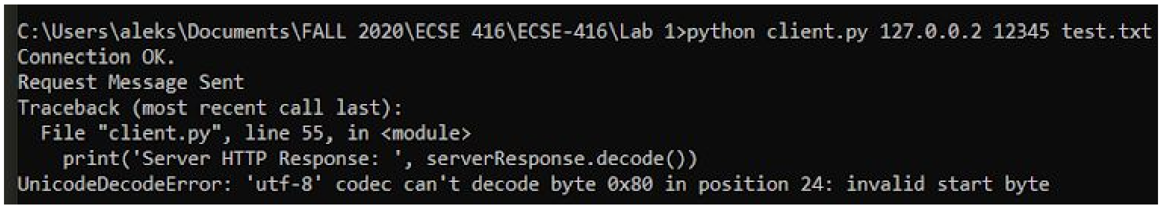
Experiment 2

When the socket.close() is commented out in the server python file and the client runs several times, the socket timeout goes into effect and terminates the program as seen in Figure 4.c. We then tested what would occur when the socket does not close on the client side. In this case, the client is unable to read one of the responses (see Figure 4.d.)



```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 test.txt
Connection OK.
Request Message Sent
Traceback (most recent call last):
  File "client.py", line 54, in <module>
    serverResponse = clientSocket.recv(1024)
socket.timeout: timed out
```

Figure 4.c. Test when the socket does not close on the server side

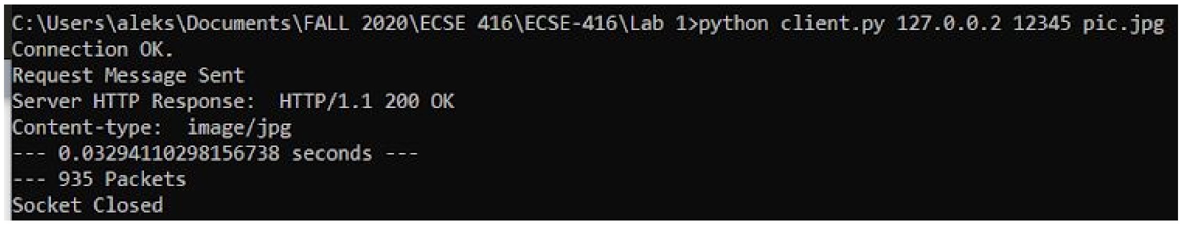


```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 test.txt
Connection OK.
Request Message Sent
Traceback (most recent call last):
  File "client.py", line 55, in <module>
    print('Server HTTP Response: ', serverResponse.decode())
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 24: invalid start byte
```

Figure 4.d Test when the socket does not close on the client side.

Experiment 3

As shown in Figure 4.e, we are recording the time and the number of packets required to process the image file. We recorded the number of packets with a simple integer that would iterate upon the retrieval of each packet, and the time was calculated by starting a time before the first message received. We tested at five times for five different packet sizes: 1024, 512, 2048, 4096 and 2 bits. The average time in microseconds of each packet sizes are 39.66, 42.94, 58.4, 24.62 and 2093.92 respectively as seen in Table 1. As we can see in Figure 4.g, increasing the chunk size significantly improves the transfer speed. In fact, there is a large difference of 15 seconds between sending chunk sizes of 1024 and 4096.



```
C:\Users\aleks\Documents\FALL 2020\ECSE 416\ECSE-416\Lab 1>python client.py 127.0.0.2 12345 pic.jpg
Connection OK.
Request Message Sent
Server HTTP Response:  HTTP/1.1 200 OK
Content-type:  image/jpg
--- 0.03294110298156738 seconds ---
--- 935 Packets
Socket Closed
```

Figure 4.e Expected output to test speed

Word Size	Packets	Average Time (µs)
2	478’245	2’093.92
512	1’869	42.94
1024	935	39.66
2048	468	58.4
4096	234	24.62

Table 1. Table of times and packets required to transmit pic.jpg

5. Discussion

Our goal throughout this lab was to create a client and a server that could communicate and share files. We developed a server that runs infinitely awaiting requests with a socket, and a client that would connect through a handshake to complete a request. In the case that the server can find a requested file, the file contents are transported to the client and displayed to the user, otherwise a HTTP 404 error is returned to the client. Through thorough testing, we ensured our code met the specifications detailed in the lab. Via the experiments, we also tested the time it took our client to transmit data dependent on packet size, in which we found that larger packets (fewer send/receives) resulted in much faster transfers. Our largest challenge while developing our lab was finding a proper way to send files of differing sizes and types and then display them. We accomplished this by having the client receive the file contents via pickled data in an infinite loop until an empty packet was received, which suggests that the file has been fully read. The received packets are concatenated into a buffer that can be read by the client. Another option would have been the server reading the file size and then transmitting that to the client before the transfer, then the client could receive in one send/receive.